MassWare User's Manual

The MassWare Team
LONG Lab, Lehigh University
http://marches.cse.lehigh.edu/

May. 27, 2009

1. Introduction

1.1 MassWare

MassWare (Mobile Ad-hoc and Sensor Systems' Middleware) is a component-based context-aware adaptive middleware framework for wireless sensor networks based on SOS (a module-based sensor operating system). MassWare supported sensor applications consists of a list of components which provide some interfaces, like operation interfaces and communication interfaces. MassWare can then efficiently reconfigure the behaviors of the applications through component parameter tuning and architecture reconfiguration at run-time. Further more, MassWare use the event-notification model for communication and context detection to overlap the reconfiguration behavior with other sensor activities. The middleware and some supported applications have been implemented and evaluated in our test-bed based MicaZ motes.

Compared to the traditional middleware that supports the single component-chain based application architecture, MassWare maintains multiple component chains (Fig. 1b). Therefore, there is a new method proposed for the behavior reconfiguration that switches active and inactive chains. This new method replaces the traditional method of modifying the single-chain structure to reduce the local behavior change time. Further, according to the new method, an efficient active-message based asynchronous synchronization protocol is proposed for synchronizing the behaviors of distributed programs. This results in a dramatic reduction of the distributed behavior synchronization time by eliminating the operation suspension time and buffer clearance time. The robustness of the distributed application is improved since the use of active messages results in no synchronous communication and system halting in the synchronization process.

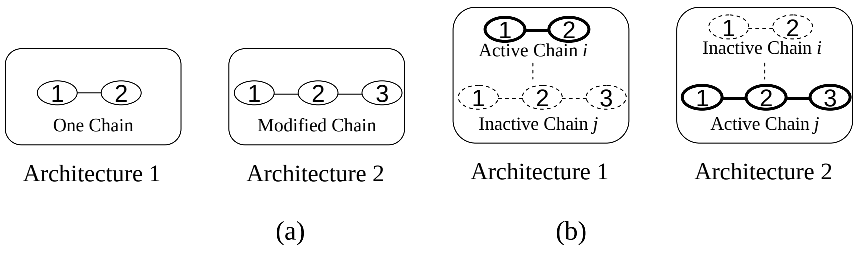


Fig. 1. Dynamic reconfiguration architecture: (a) single component-chain architecture in existing middleware, (b) multiple component-chain architecture in MassWare.

MassWare is located between the lower hardware (sensor motes) and operating system (SOS) layer and the upper application layer to monitor environments and support application adaptation. It is client-server middleware and the sensor middleware runs in the client mode and the base-station runs in the server mode. There is one middleware agent in each sensor mote. The MassWare agent is a special component that is automatically generated based on user-defined adaptation policies, which are defined by application developers or end users in a script file based on XML. The agent can then subscribed to measurement tool components, which measures application contextual information, and manipulate the computing components, which are the basic elements to construct a sensor application.

1.2 Demonstration

This demonstration illustrates the major functionality of “MassWare”: a context-aware adaptive middleware framework middleware framework that efficiently handles dynamic reconfiguration for wireless sensor applications in a generic, consistent, and clearly-separated way. By providing a set of tools and underlying mechanisms, it achieves it primary goal of facilitating a software developer to handle adaptation inside their applications, therefore to let the software developer to be more focused on application's own logic instead of dealing with the complexity of adaptation.

The Demonstration also validates the efficiency of the run-time reconfiguration for supported applications. There are two applications implemented and used in this demonstration:

Blink application: the blink application is a simple sensor application running in a single sensor mote. There are one measurement tool component (*neighboring*), which measures the number of its neighbor sensor motes, and three application components (*BlinkR*, *BlinkG*, and *BlinkY*), which control the red LED, the green LED, and the yellow LED separately. The three components are connected in a sequence, in which *BlinkR* is the starting component that blinks the red LED repetitively triggered by a timer and sends a message through its output interface to the connected component each time when it blinks. The *BlinkG* and *BlinkY* components will blink the LED each time when they receive a message from their input interfaces and send a message through their output interface to the connected components after they blink. The application architecture can be dynamically reconfigured by connecting three components (*BlinkR* → *BlinkG* → *BlinkY*) or the components (*BlinkR* → *BlinkY*) and changing the blink frequency of the *BlinkR* component (blink once every 3 sec in the 3 component architecture and every 2 sec in the 2 component architecture) based on the number of its neighbors.
Sensor data compression application: The sensor data compression application is a client-server application for sensor data compression. The sensor (client) program can sample sensor readings, compress the sensor data, and send the compressed data to the base station (server). The base station program then decompresses and displays the received data. A compression algorithm normally consists of three steps: translation, quantization, and coding. Specifically, we use Lifting Scheme Wavelet Transform (LSWT), scalar quantization, and Distributed Source Coding (DSC) or unary coding to compress the sample sensor data. There are one measurement tool component (*neighboring*), which measures the number of its neighbor sensor motes and signal strength, and five application components (*Sampling*, *LSWT*, *Quantz*, *DSC*, and *Unary*), which sample the data and perform the LSWT, the scalar quantization, the DSC, and the unary coding separately. The sensor application architecture can be dynamically reconfigured by using the DSC component (*Sampling* → *LSWT* → *Quantz* → *DSC*) or the unary coding components (*Sampling* → *LSWT* → *Quantz* → *Unary*) based on the number of its neighbors and their signal strength. The base station application architecture will also be reconfigured at runtime by using the DSC decoding component or the unary decoding component to process the received compressed data correctly.

2. Installation Instructions

- **Requirements:**

  Operating System: Windows XP and Ubuntu 6.06

  Memory: 256MB (minimum) or more

  Development kit: Microsoft Visual Studio 2005 with .NET 2.0 and SOS (Sensor Operating System)

  Other required tools: MSXML 4.0

- **Provided file list for the demo applications:**

MassCompiler\MassGen.exe	// the compile that generates the middleware source file based on XML script file
MassWare\Makefile	// the make file that generate a middleware (massware) component
MassWare\massware.h	// the header file of massware
MassWare\blink\massware.c	// the massware source file of the blink application that automatically generated // based policy_blink.xml
MassWare\compress\massware.c	// the massware source file of the compression application that automatically // generated based policy_compress.xml

- **Provided example components:**

  1) Measurement Tool Components:

Neighbor\Makefile	// the make file of the neighbor component that gets the number of neighbors
Neighbor\neighbor.h	// the header file of the neighbor component that gets the number of neighbors
Neighbor\neighbor.c	// the source file of the neighbor component that gets the number of neighbors

  * Users can use their own developed measurement tools by just modifying the tool definition in the script file

  2) Reconfigurable Components:

  a) Bink application

BlinkR\Makefile	// the make file of the blinkr component that blinks the red LED
BlinkR\blinkr.h	// the header file of the blinkr component that blinks the red LED
BlinkR\blinkr.c	// the source file of the blinkr component that blinks the red LED
BlinkG\Makefile	// the make file of the blinkg component that blinks the green LED
BlinkG\blinkg.h	// the header file of the blinkg component that blinks the green LED
BlinkG\blinkg.c	// the source file of the blinkg component that blinks the green LED
BlinkB\Makefile	// the make file of the blinkb component that blinks the yellow LED
BlinkB\blinkb.h	// the header file of the blinkb component that blinks the yellow LED
BlinkB\blinkb.c	// the source file of the blinkb component that blinks the yellow LED

  b) Compression application

Sensing\Makefile	// the make file of the sensing component that senses light density (photo)
Sensing \sensing.h	// the header file of the sensing component that senses light density (photo)
Sensing \sensing.c	// the source file of the sensing component that senses light density (photo)
LSWT\Makefile	// the make file of the lswt component that transfers sampled data based on the // Lifting Scheme Wavelet Transfer algorithm
LSWT\lswt.h	// the header file of the lswt component that transfers sampled data based on the // Lifting Scheme Wavelet Transfer algorithm

LSWT\lswt.c	// the source file of the lswt component that transfers sampled data based on the // Lifting Scheme Wavelet Transfer algorithm
Quantz\Makefile	// the make file of the quantz component that quantizes transferred data based on the // scalar quantization algorithm
Quantz\quantz.h	// the header file of the quantz component that quantizes transferred data based on the // scalar quantization algorithm
Quantz\quantz.c	// the source file of the quantz component that quantizes transferred data based on the // scalar quantization algorithm
DSC\Makefile	// the make file of the dsc component that encodes quantized data based on the // Distributed Source Coding algorithm
DSC\dsc.h	// the header file of the dsc component that encodes quantized data based on the // Distributed Source Coding algorithm
DSC\dsc.c	// the source file of the dsc component that encodes quantized data based on the // Distributed Source Coding algorithm
Unary\Makefile	// the make file of the unary component that encodes quantized data based on the // modified unary coding algorithm
Unary\unary.h	// the header file of the unary component that encodes quantized data based on the // modified unary coding algorithm
Unary\unary.c	// the source file of the unary component that encodes quantized data based on the // modified unary coding algorithm

* Users can also use their own components, and please refer the XML file description in the following pages.

Provided XML script files

policy.xsd	// XML schema file, any user developed policy file should follow this schema.
policy_blink.xml	// XML-based adaptation-policy script file for the blink application
policy_compress.xml	// XML-based adaptation-policy script file for the compression application

3. Usage Instructions

3.1 Generate MassWare Source File

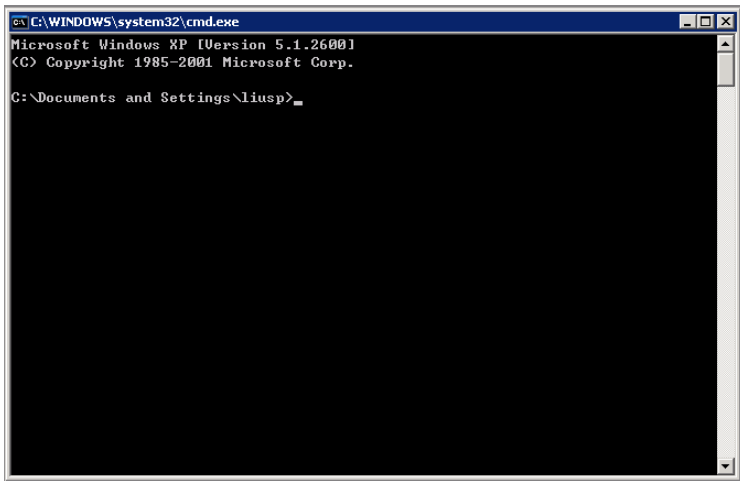
The MassWare compiler, which compiles user-defined XML adaptation policy files and generates MassWare component source file automatically, is developed in Windows system with C# language in Visual studio 2005. To run the compiler (MassGen.exe), we have to setup its running environment. (The compiler will be migrated to Ubuntu Linux system with C/C++ language to be integrated with the SOS environment.)

Operating Systems: Windows XP professional with service pack 3

Pre-installed software: .NET 2.0 framework and MSXML 4.0

Execution steps:

1) Open a command window (click *start* -> *run…* , input *cmd*)



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\liusp>_
```

2) Change the directory to the compiler folder, and put the adaptation policy file in the same folder

cd C:\cygwin\home\liusp\sos-2x\modules\MassWare\MassCompiler

3) Run the compiler with the policy file as the input parameter

MassGen.exe policy_blink.xml

4) Copy the generated massware.c file to the massware component folder

cp massware.c ..\MassWare

Note: massware.c is a middle-product of the middleware component as we need to use the SOS compiler to compile the massware.c source file to generate the binary component image, which can be burned to sensor motes. If we have a C/C++ compiler in the Linux environment in the future, we should modify the Makefile by adding the compilation step. Therefore, we can make the XML script file and generate the binary component image in one step. The middleware-product (massware.c) should be deleted and hidden from end-users.

3.2 Compile Application and Middleware Components

MassWare is a software layer built above the SOS (Sensor Operating System) and we use SOS compiler to generate MassWare components. Currently, we can only successfully run SOS in Ubuntu 6.06 operating system. It fails in Cygwin environment because the upgraded Cygwin C++ compiler is not compatible with the SOS source codes. Therefore, to compile application components and middleware components, we have the following steps:

1) Install Ubuntu 6.06

   Open a Terminal from the menu Applications → Accessories → Terminal and type:

   ```
   sudo apt-get install build-essential              // build essentail
   sudo apt-get install linux-headers-$(uname -r) // install linux headers to /usr/src
   ```

2) Install SOS in Ubuntu 6.06

   Following the tutorial of SOS and the installation page:

   https://projects.nesl.ucla.edu/public/sos-2x/doc/tutorial/installation.html#linux

   Note: the mote we use is micaz instead of mica2, the board is mib510, serial port is ttyS0,

   Some often used commands:

   a) *compile component*:

   ```
    make micaz
   ```

   b) *burn the core module to sensor motes*:

   ```
   make micaz install PROG=mib510 PORT=/dev/ttyS0 ADDRESS=1
   ```

   c) *start the server*:

   ```
    ./sossrv.exe -s /dev/ttyS0
   ```

   d) *transfer a component image to a sensor mote*

   ```
   ./sos_tool.exe --insmod ../../modules/MassWare/BlinkR/blinkr.mlf
   ```

3.2 Start the application.

In this section, we will learn how to run a blink application and a sensor data compression application step by step. Before we run the application, we need to setup the hardware:

1) Attach a MicaZ mote to a Mib510 programming board
2) Connect the Mib510 board to the PC that was installed the Ubuntu and SOS through the serial port
3) Connect the Mib510 power adapter and turn on the programming board

Bink application:

1) Generate the middleware component source file of the blink application in the windows system based on its policy file: policy_blink.xml

   ```
    MassGen.exe policy_blink.xml
   ```

2) Copy the generated file to the MassWare folder of the Ubuntu system

   ```
   cp massware.c $SOSROOT/modules/MassWare/MassWare/
   ```

3) Compile all the components: including measurement tool components, application computing components, and middleware components.

   ```
    cd $SOSROOT/modules/MassWare/Neighbor/
    make micaz
    cd $SOSROOT/modules/MassWare/BlinkR/
    make micaz
    cd $SOSROOT/modules/MassWare/BlinkG/
    make micaz
    cd $SOSROOT/modules/MassWare/BlinkB/
    make micaz
    cd $SOSROOT/modules/MassWare/MassWare/
    make micaz
   ```

4) Install SOS kernel to the sensor mote (this will erase any program and components in the sensor mote).

```
cd $SOSROOT/config/blank/
make micaz install PROG=mib510 PORT=/dev/ttyS0 ADDRESS=1
```

5) Build and start the SOS server to communicate with the kernel in the sensor mote

```
cd $SOSROOT/tools/sos_server/bin/
make x86
./sossrv.exe -s /dev/ttyS0
```

6) Build the SOS tool to insert modules

```
cd $SOSROOT/tools/sos_tool/
make emu
```

7) Insert all components to the sensor mote using the insert tool that communicates with the SOS server. Please keep in mind that the middleware component is always the last one inserted.

```
./sos_tool.exe --insmod ../../modules/MassWare/BlinkR/blinkr.mlf
./sos_tool.exe --insmod ../../modules/MassWare/BlinkG/blinkg.mlf
./sos_tool.exe --insmod ../../modules/MassWare/BlinkB/blinkb.mlf
./sos_tool.exe --insmod ../../modules/MassWare/Neighbor/neighbor.mlf
./sos_tool.exe --insmod ../../modules/MassWare/MassWare/massware.mlf
```

8) We can see that all the LEDs will be blinked after the last component (the middleware component) is inserted. Based on the measured number of neighbors, there may be two or three LEDs blinking. The application architecture is reconfigured by using or not using the green LED component, and the blink interval of the red LED component is reconfigured to be 3 or 2 seconds correspondingly at runtime.

Sensor data compression application:

1) Generate the middleware component source file of the blink application in the windows system based on its policy file: policy_compress.xml

```
MassGen.exe policy_compress.xml
```

2) Copy the generated file to the MassWare folder of the Ubuntu system

```
cp massware.c $SOSROOT/modules/MassWare/MassWare/
```

3) Compile all the components: including measurement tool components, application computing components, and middleware components.

```
cd $SOSROOT/modules/MassWare/Neighbor/
make micaz
cd $SOSROOT/modules/MassWare/Sensing/
make micaz
cd $SOSROOT/modules/MassWare/LSWT/
make micaz
cd $SOSROOT/modules/MassWare/Quantz/
make micaz
cd $SOSROOT/modules/MassWare/DSC/
make micaz
cd $SOSROOT/modules/MassWare/Unary/
make micaz
cd $SOSROOT/modules/MassWare/MassWare/
make micaz
```

4) Install SOS kernel to the sensor mote (this will erase any program and components in the sensor mote).

```
cd $SOSROOT/config/blank/
make micaz install PROG=mib510 PORT=/dev/ttyS0 ADDRESS=1
```

5) Build and start the SOS server to communicate with the kernel in the sensor mote

```
cd $SOSROOT/tools/sos_server/bin/
make x86
./sossrv.exe -s /dev/ttyS0
```

6) Build the SOS tool to insert modules

```
cd $SOSROOT/tools/sos_tool/
make emu
```

7) Insert all components to the sensor mote using the insert tool that communicates with the SOS server. Please keep in mind that the middleware component is always the last one inserted.

```
./sos_tool.exe --insmod ../../modules/MassWare/ Sensing/sensing.mlf
./sos_tool.exe --insmod ../../modules/MassWare/LSWT/lswt.mlf
./sos_tool.exe --insmod ../../modules/MassWare/Quantz/quantz.mlf
./sos_tool.exe --insmod ../../modules/MassWare/DSC/dsc.mlf
./sos_tool.exe --insmod ../../modules/MassWare/Unary/unary.mlf
./sos_tool.exe --insmod ../../modules/MassWare/Neighbor/neighbor.mlf
./sos_tool.exe --insmod ../../modules/MassWare/MassWare/massware.mlf
```

8) The application will run after the last component (the middleware component) is inserted. Based on the measured number of neighbors, The application architecture is reconfigured by using the DSC component or Unary component at runtime.

4. Development Instructions

4.1 The MassWare system UML diagram

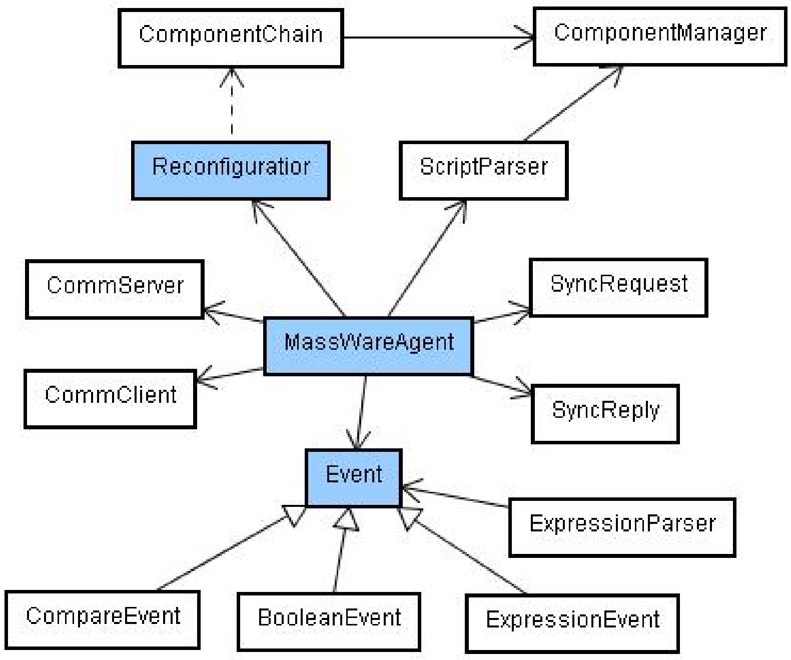


Figure 6. The UML diagram of MassWare Implementation

As shown in Fig. 6, MassWare contains three core classes (blue color): *Event*, *MassWareAgent*, and *Reconfigurator*. The *Event* class is the core class of MassWare sensor, which can be further inherited by *CompareEvent*, *BooleanEvent*, and *ExpressionEvent*. The *CompareEvent* contains two double-type event sources (left hand side and right hand side) and a compare conditioner for comparing the two event sources. The *BooleanEvent* contains two Boolean-type event sources and a Boolean conditioner. The *ExpressionEvent* is the class to build context expressions in a binary tree manner. It supports addition, subtract, multiply, and division operations and brackets. The *ExpressionParser* is used to parse the *ExpressionEvent* to build the expression tree.

The *MassWareAgent* class is the core class of the MassWare decision engine. It invokes the *ScriptParser* object to parse the user defined adaptation policy script, builds the hierarchical events, and subscribes the actuators to corresponding event sensors. It also communicates with peer agents for synchronization after reconfiguration.

The *Reconfigurator* is the core class of the MassWare reconfigurator. It contains multiple actuators and each actuator contains a component chain. The corresponding actuator notified by the context sensor will active its component chain to process the application data.

4.2 MassWare application development

All the source code are implemented in Microsoft Visual Studio 2005

1. Set the environment:

Add the middleware reference to the application project: MassWare.dll

2. Define the middleware object and instantiate it with the script file

```
MassWare.MassWare thisWare = null;
thisWare = new MassWare.MassWare(@"D:\temp\vcRules.xml");
```

3. Initialize the middleware and define the architecture change callback function:

```
thisWare.Initialize();
thisWare.AddArchSubscriber(new MassWare.Architecture.MassWareArchChangeEventHandler(MassWareArchChanged));
```

4. Set the application level component parameters

```
// set the parameter list
object[] paramlist = { localWnd.Handle, localWnd.Width, localWnd.Height };
// set the parameter for the component “Masslets.WebCam.WebCam”
thisWare.SetAppParameter("Masslets.WebCam.WebCam", "SetLocalWindowInf", paramlist);
paramlist = new object[3] { remoteWnd.Handle, remoteWnd.Width, remoteWnd.Height };
thisWare.SetAppParameter("Masslets.Display.Display", "SetDisplayInf", paramlist);
thisWare.SetAppParameter("MassTools.ABWPTR.ABWPTR", "SetRemoteAddr", new object[] { remoteIP });
```

5. Start Engine

```
 // start the middleware and run-time reconfigure the application behavior accroding to the adaptation plicies.
 thisWare.Start();
```

6. Implement the behavior change call back function.

```
private void MassWareArchChanged(object sender, MassWare.MassWareArchEventArgs e)
{
     if (e.mType == MassWare.MassWareArchEventArgs.PROACTIVE_CHANGE)
     {
          toolStripStatusLabel.Text = "Proactive Architecture Changed: " + e.mMessage;
     }
     else if (e.mType == MassWare.MassWareArchEventArgs.REACTIVE_CHANGE)
     {
          toolStripStatusLabel.Text = "Reactive Architecture Changed: " + e.mMessage;
     }
 }
```

7. Stop and release the middleware object

```
if (thisWare!= null)
{
     thisWare.Stop();
     thisWare.Dispose();
     GC.Collect();
 }
```

5. Source File List:

MassWare\Makefile	// the make file that generate a middleware (massware) component
MassWare\massware.h	// the header file of massware
MassWare\blink\massware.c	// the massware source file of the blink application that automatically generated
	// based policy_blink.xml
MassWare\compress\massware.c	// the massware source file of the compression application that automatically
	// generated based policy_compress.xml
Neighbor\Makefile	// the make file of the neighbor component that gets the number of neighbors
Neighbor\neighbor.h	// the header file of the neighbor component that gets the number of neighbors
Neighbor\neighbor.c	// the source file of the neighbor component that gets the number of neighbors
BlinkR\Makefile	// the make file of the blinkr component that blinks the red LED
BlinkR\blinkr.h	// the header file of the blinkr component that blinks the red LED
BlinkR\blinkr.c	// the source file of the blinkr component that blinks the red LED
BlinkG\Makefile	// the make file of the blinkg component that blinks the green LED
BlinkG\blinkg.h	// the header file of the blinkg component that blinks the green LED
BlinkG\blinkg.c	// the source file of the blinkg component that blinks the green LED
BlinkB\Makefile	// the make file of the blinkb component that blinks the yellow LED
BlinkB\blinkb.h	// the header file of the blinkb component that blinks the yellow LED
BlinkB\blinkb.c	// the source file of the blinkb component that blinks the yellow LED
Sensing\Makefile	// the make file of the sensing component that senses light density (photo)
Sensing \sensing.h	// the header file of the sensing component that senses light density (photo)
Sensing \sensing.c	// the source file of the sensing component that senses light density (photo)
LSWT\Makefile	// the make file of the lswt component that transfers sampled data based on the
	// Lifting Scheme Wavelet Transfer algorithm
LSWT\lswt.h	// the header file of the lswt component that transfers sampled data based on the
	// Lifting Scheme Wavelet Transfer algorithm
LSWT\lswt.c	// the source file of the lswt component that transfers sampled data based on the
	// Lifting Scheme Wavelet Transfer algorithm
Quantz\Makefile	// the make file of the quantz component that quantizes transferred data based on the
	// scalar quantization algorithm
Quantz\quantz.h	// the header file of the quantz component that quantizes transferred data based on the
	// scalar quantization algorithm
Quantz\quantz.c	// the source file of the quantz component that quantizes transferred data based on the
	// scalar quantization algorithm
DSC\Makefile	// the make file of the dsc component that encodes quantized data based on the
	// Distributed Source Coding algorithm
DSC\dsc.h	// the header file of the dsc component that encodes quantized data based on the
	// Distributed Source Coding algorithm
DSC\dsc.c	// the source file of the dsc component that encodes quantized data based on the
	// Distributed Source Coding algorithm
Unary\Makefile	// the make file of the unary component that encodes quantized data based on the
	// modified unary coding algorithm

Unary\unary.h	// the header file of the unary component that encodes quantized data based on the // modified unary coding algorithm
Unary\unary.c	// the source file of the unary component that encodes quantized data based on the // modified unary coding algorithm

policy.xsd	// XML schema file, any user developed policy file should follow this schema.
policy_blink.xml	// XML-based adaptation-policy script file for the blink application
policy_compress.xml	// XML-based adaptation-policy script file for the compression application

MassGen\Program.cs	// the major program file of the MassGen project
MassGen\MassParser.cs	// the class file the XML parser that parses the adaptation policy file and generates the // middleware source file
MassGen\ExprParser.cs	// the class file that parses the expression in the hierarchical event model
MassGen\MassComponent.cs	// the data structure used in the XML parser.

6. Source Code Copyright